**Swing Notes**

AWT vs Swing

|  |  |
| --- | --- |
| **AWT** | **Swing** |
| |  |  | | --- | --- | | AWT components are platform-dependent. |  | | Java swing components are platform-independent. |
| AWT components are heavyweight. (restrictions such as being always opaque) | Swing components are lightweight. |
| AWT doesn't support pluggable look and feel. | Swing supports pluggable look and feel. |
|  | Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc. |
| |  |  | | --- | --- | |  | AWT doesn't follows MVC(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view. | | Swing follows MVC. |

Swing Key Features

* Swing components are lightweight

This means Swing is written entirely in Java and does not rely on external code, making its components more efficient and flexible, as well as more consistent across systems.

* Swing supports a pluggable look and feel (PLAF)

Swing (rather than other the system) renders the components. It is also possible to separate the look and feel of the component from its logic. This means it is possible to “plug in” a new look and feel to a component without altering any other aspect, and unique look and feels can be defined.

The MVC Connection

A visual component contains 3 aspects:

1. how the component looks
2. how the component reacts to the user
3. the state information associated with the component
   1. example: indicator for whether a checkbox is checked or not

All component arichtectures must contain these aspects. The **model-view-controller (MVC)** is one component architecture.

In MVC, the model corresponds to the state information associated with the component; the view determines how the component is displayed (look); the controller determines how the component reacts to user input (feel).

By dividing the aspects, each can be changed without altering the others.

Swing uses a modified version of MVC that combines view and controller into the **UI delegate**. This model is called the model-delegate.

Most Swing components contain two objects: one represents the model; the other represents the delegate. Models are defined by interfaces, e.g. *ButtonModel*; UI delegates are classes that inherit *ComponentUI*, e.g. *ButtonUI*.

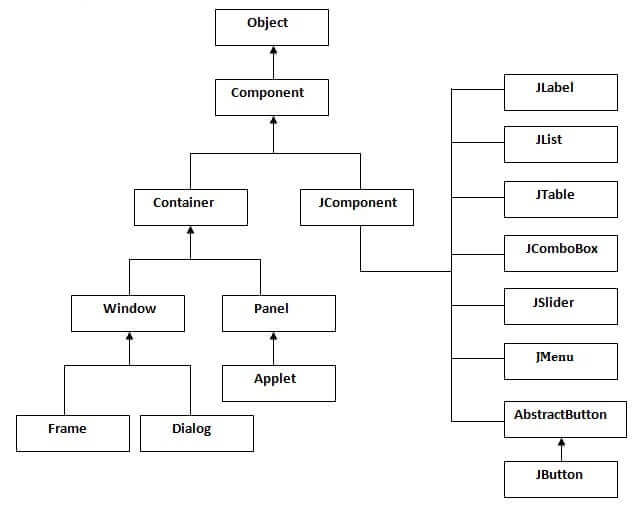
**Components and Containers**

A Swing GUI consists of two items: components and containers. However, all containers are also components. They differ on intended purpose: a component is an independent visual control; a container holds a group of components (including containers).

In order for a component to be displayed, it must be held inside a container.

Swing defines a **containment hierarchy**; at the top is the **top-level container**.

Swing class hierarchy:



**Components**

https://docs.oracle.com/javase/tutorial/uiswing/components/

Most Swing components are derived from the *Jcomponent* class. The only exceptions are the top level containers. *Jcomponent* inherits the AWT classes *Container* and *Component*. All Swing components are defined within the package *javax.swing*.

Oracle how-tos: <https://docs.oracle.com/javase/tutorial/uiswing/components/componentlist.html>

|  |  |  |
| --- | --- | --- |
| Purpose | Class name | Description |
|  | *JPanel* | General purpose container for lightweight components. |
| Displaying information | *JLabel* | A component for placing unselectable text and images in a container. |
| *ImageIcon* | paints Icons from Images |
| Displaying hierarchical information | *JTree* | Displays a set of hierarchical data as an outline. The tree is presented vertically, with each entry called a node. |
| *JTable* | Displays and edits two-dimensional tables. |
| Dialog pop-up | *JDialog* | Main class for creating a custom dialog window. (commonly used for warnings) |
| *JOptionPane* | Provides a set of standard dialog boxes that prompt users for a value or informs them. |
| Styled text | *JEditorPane* |  |
| User input - text | *JTextArea* | A text component that allows editing of a multiple lines of text.  Can be set to be uneditable, then it only displays text. |
| *JTextField* | A text component that allows for the editing of a single line of text. |
| *JPasswordField* | A text component specialized for password entry. |
| User input - buttons | *JButton* | Creates a labeled button. (Like *JLabel*, but interactive) |
| *JCheckBox* | A graphical component that can be in either an on (true) or off (false) state. |
| *JRadioButton* | A graphical component that can be in either an on (true) or off (false) state. in a group. |
| User input – selection from a range | *JList* | Presents the user with a list of items, in one or more columns, to choose from. Often put in scroll panes. |
| *JComboBox* | Creates a drop down list of values for the user to choose from. Has two forms: uneditable, and editable, where the user can type in the text field. |
| *JSpinner* | A single line input field that lets the user select a number or an object value from an ordered sequence. |
| *JSlider* | Lets the user graphically select a value by sliding a knob within a bounded interval. |
| *JScrollbar* | Represents a scroll bar component in order to enable the user to select from range of values. |
| Component management | *JTabbedPane* | Lets the user switch between a group of components by clicking on a tab. |
| *JScrollPane* | Provides a scrollable view of a lightweight component. |
| *JToolBar* | Groups several components (usually buttons) into a row or column. |
| Also supplies dialog | *JFileChooser* | A dialog window from which the user can select a file. |
| Also supplies dialog | *JColorChooser* | Enables the user choose from a color palette. |
|  | *JProgressBar* | As the task progresses towards completion, the progress bar displays the task's percentage of completion. |

Containers

There are two types of containers:

* top level containers: *JFrame*, *JApplet*, *JWindow*, *JDialog*.

They do not inherit *JComponent*, but do inherit *Component* and *Container*. Top level containers are heavy weight.

Top level containers must be at the top of a containment hierarchy.

The most commonly used one is *JFrame*.

(*JApplet* was deprecated when applets were deprecated in JDK 9. Applet support was removed by JDK 11.)

* lightweight containers, e.g. *JPanel*.

Panes

Each top-level container defines a set of panes.

A *JRootPane* instance is at the root of the hierarchy. The root pane is a lightweight container that manages other panes. It is comprised of the *glass pane*, the *content pane*, and the *layered pane*.

* The glass pane sits above and completely covers the other panes. It is by default a transparent *JPanel*. It manages mouse events that affect the entire container, and can paint over other compoents.
* The layered pane is an instance of *JLayeredPane*. It gives components depth values and determines which component appear on top. It holds the content pane and (optional) menu bar.
* The content pane contains the visual components. By default, it is an opaque *JPanel*.

Example

This example uses two components: *JFrame* and *JLabel*. *JLabel* creates a label to display information.

**import** javax.swing.\*;

**public** **class** LabelExample {

LabelExample() {

JFrame jf = **new** JFrame("Swing example");

//sets the title of the window

jf.setSize(250, 150);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

//terminates the program on close

//by default, the window closes the but app remains open

JLabel label = **new** JLabel("Label example");

jf.add(label);

//by default adds to center and size adjusted automatically

//use remove() to remove components

jf.setVisible(**true**);

//JFrame is invisible by default

}

**public** **static** **void** main(String[] args) {

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** LabelExample();

}

});

}

}

Calling *invokeLater()* creates an *event dispatching thread*. The main() method causes the event dispatching thread rather than the main thread to create the *LabelExample* instance.

Swing programs are event driven and thus requires to be executed on the event dispatching thread provided by Swing. This applies to the creation and updating of all Swing components. *main()* must operate by proxy by creating a *Runnable* object that executes on the event dispatching thread.

One of two methods can create the GUI code on the event dispatching thread: *invokeLater()* and *invokeAndWait()*. They are static and defined by *SwingUtilities*. Their parameters are the *Runnable* object whose *run()* will be called by the event dispatching thread. Their difference is that *invokeLater()* returns immediately whereas *invokeAndWait()* returns when *run()* does.

**Layout Manager**

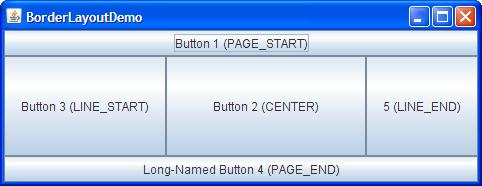
https://docs.oracle.com/javase/tutorial/uiswing/layout/using.html

Layout managers determine the sizes and positions of GUI components inside a container. A layout manager is an object that implements the *LayoutManager* interface. It is set with *setLayout()*.

The most used managers are:

* *BorderLayout*

Places components in out to 5 areas: top, bottom, left, center, right. Default for panes. Example:



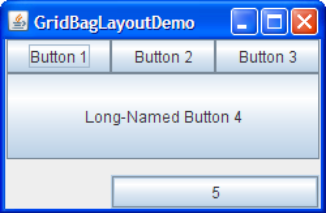
* *FlowLayout*

Places components in a single row. Default for *Jpanel*. Example:



* *GridBadLayout*

The most powerful layout manager. Aligns components with a grid of cells. Each component can span more than one cell. Example:



Other layout managers are:

* *CardLayout*

Implements an area that contains different components at different times. Controlled by a combo box, which decides which group of components to display. Example:

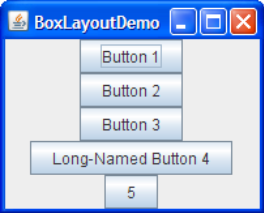


* + *JtabbedPane* is an alternative to *CardLayout*, with a predefined GUI. Example:



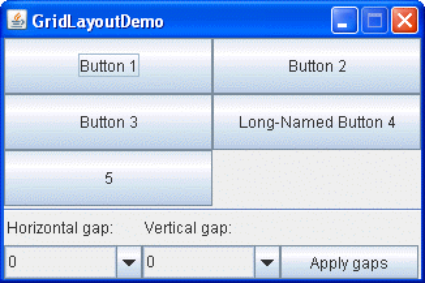
* *BoxLayout*

Places components in a single row or column. Respects components’ max sizes and can align components. Example:



* *GridLayout*

Makes the components equal in size and places them in the requested rows and columns. Example:



* *GroupLayout*

Works wih horizontal and vertical layouts separately. Each component needs to be defined twice, in each dimension.

* *SpringLayout*

Allows precise layout of components, based on relations with other components.

Size Hints

Components can provide size hints to its container’s layout manager. The component can specified its minimum, maximum, and preferred sizes. They are set with *setMinimumSize()*, *setPreferredSize()*, and *setMaximumSize()*, and obtained with *get…()*.

Many layout managers ignore requested maximum sizes, however *BoxLayout* and *SpringLayout* do not.

*GroupLayout* provides the ability to set the minimum, maximum and preferred sizes of components by itself (without the components).

Alignment Hints

Components’ alignments can be set with *setAlignmentX()*, *setAlignmentY()*, or by overriding *get…()*. Most layout managers ignore alignment hints, except *BoxLayout*.

Spacing

Each layout manager has different spacing supports.

Invisible components can be used for spacing. *BoxLayout* supports this.

Empty borders can also be added to components to produce the effect of spacing.

**Painting in Swing**

Swing lets you write directly into the display areas of components. Drawing methods are provided by the AWT (Ch.25), with some differences.

Fundamentals

AWT defines *paint()* to draw onto the surface of components. It is generally not called by programs, but rather by the runtime system when rendering components.

Swing has three methods for painting lightweight components: *paintComponent(), paintBorder(), paintChildren()*. When *paint()* is called, these methods are called in this order.

To paint the surface of a component, create a subclass of the component and override *paintComponent()*. When doing so, begin with *super.paintComponent()*, then implement the method.

To paint a component while under program control, call *repaint()*, which causes the runtime to call *paint()* as soon as possible.

Compute paintable area

If the paint goes into the border area, it will be overwritten when the border is drawn. To avoid this, restrict the paintable area to the size of the component minus the space used by the border.

The size of the border is described by an *Insets*. Obtain the border size by calling *getInsets()*. *Insets* contains the fields *top, bottom, left, right*.

Example: draw lines that are generated randomly.

**import** java.awt.Color;

**import** java.awt.Graphics;

**import** java.awt.Insets;

**import** java.util.Random;

**import** javax.swing.BorderFactory;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.JPanel;

**import** javax.swing.SwingUtilities;

**class** PaintPanel **extends** JPanel {

Insets ins;

Random rdm;

PaintPanel() {

setBorder(BorderFactory.*createLineBorder*(Color.***BLUE***, 5));

//sets the border of this element

//takes a Border as parameter

//BorderFactory provides methods that create Borders

rdm = **new** Random();

}

@Override

**protected** **void** paintComponent(Graphics g) {

//size not specified; by default sized to fill the window

//orientation not specified; by default adds to center

**super**.paintComponent(g);

**int** x,y,x2,y2,

height = getHeight(), width = getWidth();

ins = getInsets();

**for**(**int** i=0;i<10;i++) {

x = rdm.nextInt(width-ins.left);

//restrict between 0 and width-ins.left

y = rdm.nextInt(height-ins.bottom);

x2 = rdm.nextInt(width-ins.left);

y2 = rdm.nextInt(height-ins.bottom);

g.drawLine(x, y, x2, y2);

//draw line between (x,y) and (x2,y2)

}

}

}

**public** **class** PaintEx {

JLabel label;

PaintPanel pp;

PaintEx() {

JFrame jf = **new** JFrame("Paint demo");

jf.setSize(200, 200);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

pp = **new** PaintPanel();

//each time the window is resized or minimized, the lines are redrawn

jf.add(pp);

jf.setVisible(**true**);

}

**public** **static** **void** main(String[] args) {

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** PaintEx();

}

});

}

}

**Event Handling**

Swing uses the same event handling mechanism as AWT.

Example:

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** LabelExample {

JLabel label = **new** JLabel("Press a button");

LabelExample() {

JFrame jf = **new** JFrame("Event example");

jf.setSize(250, 150);

jf.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

jf.setLayout(**new** FlowLayout()); //ch26

//overrides the default, which is BorderLayout

jf.setVisible(**true**);

JButton btnA = **new** JButton("A");

JButton btnB = **new** JButton("B");

btnA.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent ae) {

label.setText("A");

}

});

//the ActionListener interface defines only one method //actionPerformed(), which is called on an action.

/\*can also use lambda expression:

\* btnA.addActionListener((ae) -> label.setText(“A”)); \*/

btnB.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent ae) {

label.setText("B");

}

});

jf.add(label);

jf.add(btnA);

jf.add(btnB);

}

**public** **static** **void** main(String[] args) {

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** LabelExample();

}

});

}

}

All event handlers are called on the event dispatching thread. If it must do something time consuming, it should run on a separate thread.

**Swing Event Handling**

There are two types of events:

* foreground event – generated by user interaction with the GUI. For example, clicking a mouse.
* background event – not generated by the user. For example, OS interrupt, software failure.

Java uses the delegation event model to handle events. This model has two components:

* source – the object on which the event occurs, for example a button. Provides information to the handler.
* listener/handler – generates a response to the event. Also an object. Needs to register with the source.

This model separates the UI logic from the logic that generates the event.

Steps to event handling:

1. the event occurs
2. an instance of the concerned *Event* is created with the source information
3. the *Event* object is forwarded to the registered listener
4. the listener *actionPerformed()* method runs

Writing an action listener

<https://docs.oracle.com/javase/tutorial/uiswing/events/index.html>

Action listeners are the easiest and most common event handlers to implement. To write an action listener, follow the following steps:

1. declare a class that implements *ActionListener* or extends an implementation of *ActionListener*.
2. implement *actionPerformed(ActionEvent ae)* in the class
   1. *actionPerformed()* is *ActionListener*’s only defined method
3. register an instance of the event handler class on the source component(s), e.g.

srcComponent.addActionListener(new myListener());

Example:

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** javax.swing.\*;

**public** **class** SwingControlDemo {

**private** JFrame mainFrame;

**private** JLabel headerLabel, statusLabel;

**private** JPanel controlPanel;

**public** SwingControlDemo() {

prepareGUI();

}

**public** **static** **void** main(String[] args) {

SwingControlDemo swingControlDemo = **new** SwingControlDemo();

swingControlDemo.showEventDemo();

}

**private** **void** prepareGUI() {

mainFrame = **new** JFrame("Java SWING Examples");

mainFrame.setSize(400,400);

mainFrame.setLayout(**new** GridLayout(3, 1));

headerLabel = **new** JLabel("",JLabel.***CENTER*** );

statusLabel = **new** JLabel("",JLabel.***CENTER***);

statusLabel.setSize(350,100);

mainFrame.addWindowListener(**new** WindowAdapter() {

**public** **void** windowClosing(WindowEvent windowEvent) {

System.*exit*(0);

}

});

controlPanel = **new** JPanel();

controlPanel.setLayout(**new** FlowLayout());

mainFrame.add(headerLabel);

mainFrame.add(controlPanel);

mainFrame.add(statusLabel);

mainFrame.setVisible(**true**);

}

**private** **void** showEventDemo() {

headerLabel.setText("Control in action: Button");

JButton okButton = **new** JButton("OK");

JButton submitButton = **new** JButton("Submit");

JButton cancelButton = **new** JButton("Cancel");

okButton.setActionCommand("OK");

submitButton.setActionCommand("Submit");

cancelButton.setActionCommand("Cancel");

okButton.addActionListener(**new** ButtonClickListener());

submitButton.addActionListener(**new** ButtonClickListener());

cancelButton.addActionListener(**new** ButtonClickListener());

controlPanel.add(okButton);

controlPanel.add(submitButton);

controlPanel.add(cancelButton);

mainFrame.setVisible(**true**);

}

**private** **class** ButtonClickListener **implements** ActionListener {

**public** **void** actionPerformed(ActionEvent e) {

String command = e.getActionCommand();

**if**( command.equals( "OK" )) {

statusLabel.setText("Ok Button clicked.");

} **else** **if**( command.equals( "Submit" ) ) {

statusLabel.setText("Submit Button clicked.");

} **else** {

statusLabel.setText("Cancel Button clicked.");

}

}

}

}

Event Objects

All event state objects derive from *EventObject*.

Events are constructed with a reference to the source.